

CS 101: Computer Programming and Utilization

Puru

with

CS101 TAs and Staff

Course webpage: <https://www.cse.iitb.ac.in/~cs101/>

Lecture 23: The C++ Standard Template Library

Classes

- A **class** is essentially the same as a struct, except:
 - Any members/member functions in a struct are public by default
 - Any members/member functions in a class are private by default

Classes

- Example: A Queue class

```
class Queue{  
    int elements[N], nWaiting, front;  
    public:  
        Queue() {...}  
        bool remove(int &v) {...}  
        bool insert(int v) {...}  
};
```

Queue q;

q.front=0; X

q.insert(5); ✓

$$\left\{ \begin{array}{l} \text{front} = (\text{front} + 1) \% N; \end{array} \right.$$

- The members - `elements`, `nWaiting` and `front` will be private.

Function definition and declaration (with class)

```
class V3{
    double x,y,z;
    V3(double v){
        x = y = z = v;
    }
    double X(){
        return x;
    }
};
```

```
class V3{
    double x,y,z;
    V3(double v);
    double X();
};

//implementations
V3::V3(double v){
    x = y = z = v;
}
double V3::X(){
    return x;
}
```

Example (with struct)

```
struct V3{  
    double x,y,z;  
    V3(double v){  
        x = y = z = v;  
    }  
    double X(){  
        return x;  
    }  
};
```

```
struct V3{  
    double x,y,z;  
    V3(double v);  
    double X();  
};  
  
//implementations  
V3::V3(double v){  
    x = y = z = v;  
}  
double V3::X(){  
    return x;  
}
```

The C++ Standard (Template) Library

Chapter 22

The C++ Standard Library

- Comes with every C++ distribution
- Contains many functions and classes that you are likely to need in day to day programming
- The classes have been optimized and debugged thoroughly
- If you use them, you may be able to write programs with very little work
- Highly recommended that you use functions and classes from the standard library whenever possible
- `Files, Strings, Maps, Vectors, Sets, Lists, Queues ...`

Input Output Classes (stdin/stdout/files)

variables/objects

classes/struct definitions

- cin, cout : objects of class istream, ostream resp. predefined in C++
- <<, >> : operators defined for the objects of these classes
- ifstream: another class like istream
- You create an object of class ifstream and associate it with a **file** on your computer
- Now you can read from that file by invoking the >> operator!
- ofstream: a class like ostream, to be used for writing to files
- Must include header file <fstream> to use ifstream and ofstream

Example of File i/o

```
#include <fstream>
#include <simplecpp>
int main(){
    ifstream infile("f1.txt");
    // constructor call.
    // object infile is created and associated
    // with f1.txt, which must be present in the current directory

    ofstream outfile("f2.txt");
    // constructor call. Object outfile is created and associated
    // with f2.txt, which will get created in the current directory

    repeat(10){
        int v;
        infile >> v;
        outfile << v;
    }
    // f1.txt must begin with 10 numbers. These will be read and
    // written to file f2.txt
}
```

fstream

cin >> | | iostream
cout << | |

ifstream / ofstream member functions

- `open`, `close`, `is_open`
- `>>`, `<<`, `!`
- `get`, `getline`, `peek`, `read`
- `put`, `write`

“String theory”

str

- Iterative computations are demonstrated well on arrays
- **strings** ... the *system* manages the array space for us

Char str[6];

- string message; // a character string
- Can assign and append to strings
- Can read a position: cout << message[px]
- Can write a position: message[px] = 'q'

Strings without `string`

- character arrays!

```
char str[5] = { 'h', 'e', 'l', 'l', 'o' } ;
```

`cout << str;`

- why `string` then?

- the dreaded **NULL** character
- null character => character with ASCII value 0
- require ***null-terminated*** character array to represent end of string
- no end of string can lead to chaos!

`'\0'`



Challenge with char arrays!

```
char str[6] = { 'e', 'a', 'r', 't', 'h' } ;  
  
cout << str;    // early takeoff of space shuttle  
                  w/o null-termination.  
str[5] = '\0';  
cout << str;    // back to earth! ✓
```

- Have to ensure null-termination at all points
- Character array sizing has to be managed (via copies etc.) explicitly
- Not objects!

the string class

```
string str = "earth";  
cout << str; // stay on earth!
```

- can use indexing as in arrays
- other member functions

- size, clear, empty,
- +, =, +=, >>, <<
- push_back, pop_back, append
- insert, erase, find, substr

a = a + b;

||

Printing a string in reverse

```
string message;  
getline(cin, message);  
int mx = message.size()-1;  
while (mx >= 0) {  
    cout << message[mx];  
    --mx;  
}
```

ENTER
↑
get a line of i/p
from kdb

Character at
position mx in
string message

- **mx** updated in a predictable way
- Ideal candidate to write as for loop

Finding needles in a haystack

- Given two strings, needles and haystack
- needles has no repeated characters
- haystack may repeat characters
- How many characters in needles appear in haystack at least once?
- needles = “bat”, haystack = “tabla” \rightarrow 3
- needles = “tab”, haystack = “bottle” \rightarrow 2

One needle in a haystack

- Subproblem: given one character `ch` and a string find if `ch` appears in string at least once

```
char ch;  
cin >> ch;  
string haystack;  
cin >> haystack;  
  
int ans = 0; // will change to 1 if found  
for (int hx = 0; hx < haystack.size(); ++hx) {  
    if (ch == haystack[hx]) {  
        ++ans;  
        break; // quit on first match  
    }  
}
```

Many needles: nested loop

```
main() {
    string needles, haystack;
    getline(cin, needles); getline(cin, haystack);

    int ans = 0;
    for (int nx=0; nx < needles.size(); ++nx) {
        char ch = needles[nx];
        for (int hx = 0; hx < haystack.size(); ++hx) {
            if (ch == haystack[hx]) {
                ++ans;
                break; // quit on first match
            }
        } // ends haystack loop
    } // ends needles loop
}
```

**Generalize to work in
case needles can also
have repeated
characters**

Duplicate needles

- needles = “bat”, haystack = “tabla” → 3
- needles = “tab”, haystack = “bottle” → 2
- needles = “bata”, haystack = “tabla” → 3
- Two approaches
 - Dedup needles before executing earlier code (reducing to known problem)
 - Dedup needles “on the fly” (inside the nx loop)

Strings and member functions (example)

```
#include <string>
string v = "abcdab";
string w(v);
```

operator =
String s = v.substr(1,3);
CC

```
v[2] = v[3]; // indexing allowed. v becomes "abddab"
```

overload [] operator.

```
cout << v.substr(2) << v.substr(1,3) << endl;
```

// substring starting at v[2] ("ddab")

// substring starting at v[1] of length 3 ("bdd")

abddab
↑

bdd

```
int i = v.find("ab"); // find occurrence of "ab" in v
// and return index
```

```
int j = v.find("ab", 1); // find from index 1
```

```
cout << i << ", " << j << endl; // will print out 0, 4.
```

Dynamic memory allocation

- Heap memory

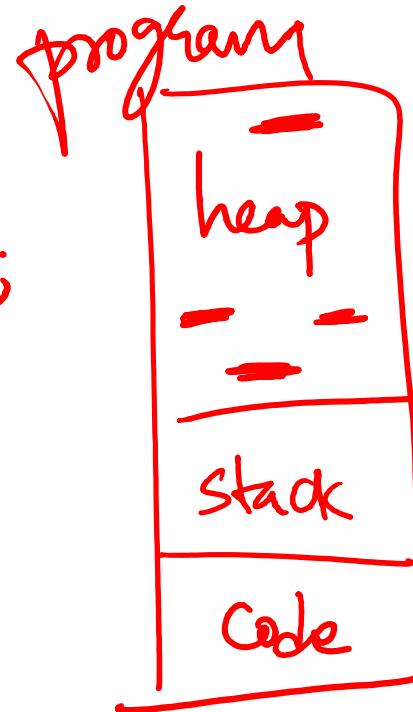
- new
- delete

- Memory leaks

`char * c = new char[10];`
`c[0] c[1] . . . c[9]`

`int a;`
`char c[5];`

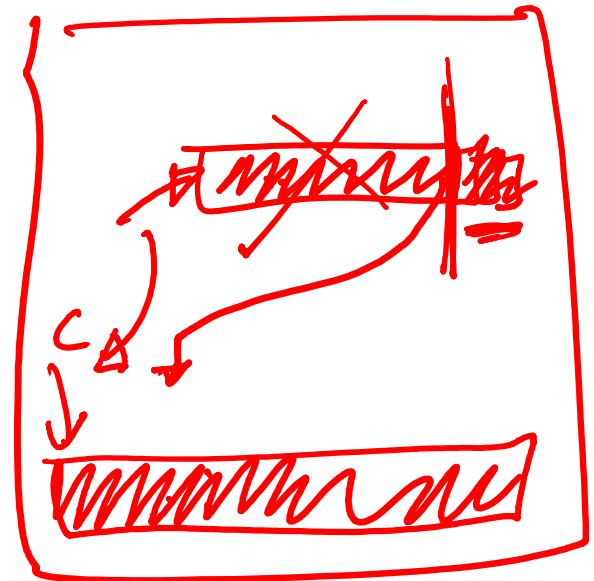
Stack
memory
activation
area.



memory leak.
int *b = - - - - ;
int *a = new int[5];
a = b;

```
class class string {  
    char *c;  
  
    string() {  
        c = new char[10];  
    }  
    resize() {  
    },  
};
```

delete c; → free allocated memory.



Queue *q = new Queue;



`q->insert(5);`

`(*q).insert(10);`

`delete(q);`

Templates

Template functions

- Function templates (Sec 12.5 in book)
- Consider these three functions: same body, different types

```
int Abs(int x)
{
    if (x < 0)
        return -x;
    else return x;
}
```

```
float Abs(float x)
{
    if (x < 0)
        return -x;
    else return x;
}
```

```
double Abs(double x)
{
    if (x < 0)
        return -x;
    else return x;
}
```

A common template to unite them all ...

```
template<typename T>
T Abs(T x) {
    if (x < 0)
        return -x;
    else return x;
}
```


Template Class

- Like function templates, create class with templates.

```
template <class T>
class Queue {
    int front, nWaiting;
    T elements[100];
public:
    bool insert(T value)
    {...}
    bool remove(T &val)
    {...}
};
```

```
main () {
    Queue<V3> q;
    Queue<int> r;
    r.insert(10);

    v V3(1,1,1);
    q.insert(v);
}
```

of V3's

of ints
queue

Vectors

- Friendlier, more versatile version of arrays
- Must include header file `<vector>` to use it
- vectors of any type by supplying the type as an argument to the template
- Indexing possible like arrays
- Possible to extend length, or even insert in the middle

vector examples

```
#include <vector>    // needed

vector<int> v1;       //empty vector. Elements will be int
vector<float> v2;     //empty vector. Elements will be float
vector<short> v3(10); // vector of length 10.
                     // Elements are of type short

vector<char> v4(5, 'a'); // 5 elements, all 'a'
cout << v3.size() << endl; // prints vector length, 10
                          // v3.length() is same

v3[6] = 34;           // standard indexing
```

operator []

vector examples (continued)

```
#include <vector>                                // needed
...

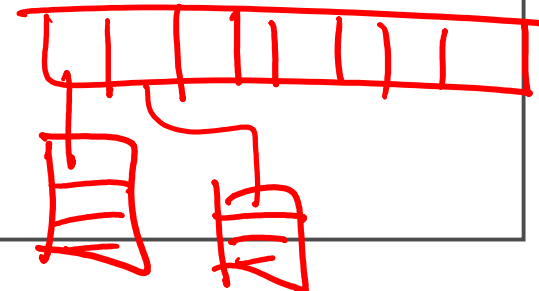
v3.push_back(22);                                // append 22 to v3.
                                                    // Length increases

vector<char> w;
w = v5;                                           // element by element copy

v1.resize(9);                                     // change length to 9
v2.resize(5, 3.3);                               // length becomes 5, all
                                                    // values become 3.3

vector<string> s;                                // vector of string

vector<vector<int> > vv;                          // allowed!
```



size_t

- The member function `size` returns a value of type `size_t`
- `size_t` is an unsigned integer type; it is meant specially for storing array indices
- When going through array elements, use `size_t` for the index variable

```
vector<double> v(10);           // initialize v
for(size_t i=0; i<v.size(); i++)
    cout << v[i] << endl;
```

- If `i` were declared `int`, then the compiler would warn about the comparison between `i` and `v.size()`
 - comparison between signed and unsigned int, which is tricky as discussed in Section 6.8.
 - By declaring `i` to be `size_t`, the warning is suppressed.

Multi-dimensional vectors

```
vector<vector<int>> > vv;  
// each element of vv is itself a vector of int  
// we must supply two indices to get to int  
// Hence it is a 2d vector!  
// Currently vv is empty
```

```
vector<vector<int>> > vv1(5, vector<int>(10, 23));  
// vv1 has 5 elements  
// each of which is a vector<int>  
// of length 10  
// having initial value 23
```

Multi-dimensional vectors usage

- Note that the syntax is not new/special
- It is merely repeated use of specifying the length and initial value:

`vector<type> name(length, value)`

- Two dimensional arrays can be accessed by supplying two indices, i.e., `vv1[4][6]` and so on
- Write `vv1.size()` and `vv1[0].size()` to get number of rows and columns

5

10

Creating a 5x5 identity matrix

```
vector<vector<double>> m(5, vector<double>(5,0));  
                        // m = 5x5 matrix of 0s  
// elements of m can be accessed  
// by specifying two indices  
for(int i=0; i<5; i++)  
    m[i][i] = 1;  
// place 1's along the diagonal
```


Ch. 22 (22.2.7)

- The book discusses a matrix class which internally uses vector of vectors
- This class is better than two dimensional arrays because it can be passed to functions by value or by reference, with the matrix size being arbitrary

Sorting a vector

- C++ provides a built-in facility to sort vectors and also arrays
- You must include `<algorithm>` to use this

```
vector<int> v(10);  
// somehow initialize v  
  
sort(v.begin(), v.end());
```

- That's it! `v` is sorted in non decreasing order
- `begin()` and `end()` return “**iterators**” over `v`.
Think of them as abstract pointers to the beginning and the end.

Sorting an array

- The algorithms in header file `<algorithm>` can also sort arrays as follows

```
double a[100];  
// somehow initialize a  
  
sort(a, a+100); // sorted!  
// second argument is name+length
```

- More variations in the book

The **Map** Template Class

Map
Sets | Lists

- A vector or an array give us an element when we supply an index
 - Index must be an **integer**
- May want to use indices which are not integers, but **strings**
 - Given the name of a country, we may want to find out its population, or its capital
 - This can be done using a **map**
 - **(a.k.a) key-value store**
 - keys and values can be of data types other than integers

The Map

- General form:

```
map<indextype, valuetype> mapname;
```

- Examples:

```
map<string, double> population;
```

Indices will have type `string` (country names), and elements will have type `double` (population)

```
map<string, vector<string>> dictionary;
```

??

Map usage

```
map<string, double> population;

population["India"] = 1.21;
// in billions. Map entry created
population["China"] = 1.35;
population["USA"] = 0.31;

cout << population["China"] << endl;
// will print 1.35

population["India"] = 1.22;
//update allowed
```

Checking index validity

```
string country;  
cout << "Give country name: ";  
cin >> country;  
  
if(population.count(country)>0) {  
    // true if element with index = country  
    // was stored earlier  
    // count is a known member function  
    cout << population[country] << endl;  
}  
else cout << "Not known.\n";
```

Remarks

- A lot goes on behind the scenes to implement a map
- Basic idea is discussed in Chapter 24 of the textbook
- ***How to print all entries of a map?***

Iterators

- A map can be thought of as holding a sequence of pairs, of the form (index, value)
- For example, the population map can be considered to be the sequence of pairs

`[("China",1.35), ("India",1.21), ("USA", 0.31)]`

- You may wish to access all elements in the map, one after another, and do something with them
- For this, you can obtain an **iterator**, which points to (in an abstract sense) elements of the sequence

Iterators (continued)

An iterator points to (in an abstract sense) elements of the sequence

- An iterator can be initialized to point to the **first** element of the sequence
- In general, given an iterator which points to some element, you can ask if there is any element **following** the element, and if so make the iterator point to the **next element**
- An iterator for a **map<index,value>** is an object with type **map<index,value>::iterator**

Using iterators

- An iterator points to elements in the map; each element is a struct with members first and second
- We can get to the members by using dereferencing
- Note that this simply means that the dereferencing operators are defined for iterators
- If many elements are stored in an iterator, they are arranged in (lexicographically) increasing order of the key

Example

```
map<string,double> population;  
population["India"] = 1.21;  
  
map<string,double>::iterator mi;  
mi = population.begin();  
// population.begin() : constant iterator  
// points to the first element of population  
// mi points to (India,1.21)  
  
cout << mi->first << endl; // or (*mi).first << endl;  
// will print out India  
  
cout << mi->second << endl;  
// will print out 1.21
```

Example

```
map<string,double> population;  
population["India"] = 1.21;  
population["China"] = 1.35;  
population["USA"] = 0.31;  
  
for(map<string,double>::iterator  
    mi = population.begin();  
    mi != population.end(); mi++)  
  
// population.end() : constant iterator  
// marking the end of population  
// ++ sets mi to point to the  
// next element of the map  
// loop body
```

Example

```
map<string,double> population;
population["India"] = 1.21;
population["USA"] = 0.31;
population["China"] = 1.35;

for (map<string,double>::iterator
    mi = population.begin();
    mi != population.end();
    mi++)
{
    cout << (*mi).first << ": " << (*mi).second << endl;
    // or cout << mi->first << ": " << mi->second << endl;
}
// will print out countries and population in
alphabetical order
```

Remarks

- Iterators can work with vectors and arrays too
- Iterators can be used to find and delete elements from maps and vectors.

```
map<string,double>::iterator  
    mi = population.find("India");  
  
population.erase(mi);
```

Maps with user-defined class as index

- Any class used as indextype on a map must implement the "<" operator.
- Example, the following code will not work because "<" is not defined on V3.
 - `class V3 {public: double x,y,z};`
 - `map<V3, string> vec2string;`
- A correct implementation of V3 may be something like:

```
class V3 {  
    public:  
    double x,y,z;  
    bool operator<(const V3& a) const {  
        if (x < a.x) return true;  
        if (x == a.x && y < a.y) return true;  
        if (x==a.x && y == a.y && z < a.z) return true;  
        return false;  
    }  
};
```


Sets

- Sets are containers that store unique elements following a specific order
- The value of the elements in a set cannot be modified once in the container (the elements are always const), but they can be inserted or removed from the container
- Internally, the elements in a set are always sorted following a specific ordering criterion indicated by its internal comparison object

Populating and Traversing a Set

```
#include <set>                // set class library

...

set<int> set1;    // create a set object,
                  // specifying its content as int
                  // the set is empty

int ar[]={3,2,4,2};

for (int i = 0; i < 4; i++) {
    set1.insert(ar[i]);    // add elements to the set.
}

for (set<int>::iterator iter = set1.begin();
     iter != set1.end(); iter++) {
    cout << *iter << " ";
}    // prints 2 3 4
```

Application of Set

Given N students where each student has a list of courses that they have taken. Create group of all students that have taken exactly the same set of courses.

```
map<set<string>, vector<int>> study_group;
// key of the map is the set of courses.
// value is vector of student roll-numbers of students
// who have taken this course.

cin >> N;

for(int i = 0; i < N; i++) {
    int roll, int n;
    cin >> roll >> n;
    set<string> subjects;
```

Application of Set (continued)

```
for (int j = 0; j < n; j++) {  
    string s; cin >> s;  
    subjects.insert(s);  
}  
study_group[subjects].push_back(rollno);  
}
```

List

- Implements a classic list data structure
- Supports a dynamic bidirectional linear list
- Unlike a C++ array, the objects the STL list contains cannot be accessed directly (i.e., by subscript)
- Is defined as a template class, meaning that it can be customized to hold objects of any type
- Responds like an unsorted list (i.e., the order of the list is not maintained).

However, there are functions available for sorting the list

Populating and Traversing a List

```
#include <list>                // list class library

...

list <int> list1;              // create a list object,
                               // specifying its content as int
                               // the list is empty

for (i=0; i<5; i++)
    list1.push_back (i);      // add at the end of the list
...
while (list1.size() > 0)
{
    cout << list1.front();    // print the front item
    list1.pop_front();        // discard the front item
}

// other functions
// insert, remove, pop_back, push_front, remove, sort, ...
```

Concluding Remarks

- Standard Library contains other useful classes, e.g. queue, list, set etc.
- The Standard Library classes use heap memory, however this happens behind the scenes and you don't have to know about it
- The library classes are very useful. Get some practice with them

More details on the web.

Example: <http://www.cplusplus.com/reference/stl/>